

# Unsatisfiable Cores and Lower Bounding for Constraint Programming

Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey

National ICT Australia\* and the University of Melbourne, Victoria, Australia  
`{ndowning,tfeydy,pjs}@csse.unimelb.edu.au`

**Abstract.** Constraint Programming (CP) solvers typically tackle optimization problems by repeatedly finding solutions to a problem while placing tighter and tighter bounds on the solution cost. This approach is somewhat naive, especially for soft-constraint optimization problems in which the soft constraints are mostly satisfied. Unsatisfiable-core approaches to solving soft constraint problems in SAT (e.g. MAXSAT) force all soft constraints to be hard initially. When solving fails they return an unsatisfiable core, as a set of soft constraints that cannot hold simultaneously. These are reverted to soft and solving continues. Since lazy clause generation solvers can also return unsatisfiable cores we can adapt this approach to constraint programming. We adapt the original MAXSAT unsatisfiable core solving approach to be usable for constraint programming and define a number of extensions. Experimental results show that our methods are beneficial on a broad class of CP-optimization benchmarks involving soft constraints, cardinality or preferences.

## 1 Introduction

Earlier work on unsatisfiable cores for Maximum Satisfiability (MAXSAT) has shown that it is advantageous to consider soft constraints to be hard constraints initially, solve the problem using a modern SAT solver, and use the resulting evidence of infeasibility to see which (temporarily hard) constraints are conflicting with each other, and soften them again only as necessary [12].

In this paper we extend the unsatisfiable cores algorithm from MAXSAT to Constraint Programming (CP). CP handles soft-constraint problems as minimization problems where the objective is a count of violations, the counts being derived from either reified primitive constraints (whose enforcement is controlled by an auxiliary variable) or soft global constraints (for examples of soft global constraints and their propagation algorithms see Van Hoes [28]).

One reason to expect that unsatisfiable cores will help in CP is that propagation solving relies on eliminating impossible (partial) solutions, but unfortunately when most constraints are soft then most solutions cannot be ruled out definitively and so propagation has little effect. Making as many constraints as

---

\* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

possible hard, should improve the propagation behaviour. Conversely, for the approach to be successful the solver needs to be able to prove infeasibility, if this is easy to do repeatedly then unsatisfiable cores will be highly effective, but if it requires a lot of search then it should be put off for as long as possible!

We work in the context of a Lazy Clause Generation (LCG) solver, because the LCG solver can easily ‘explain’ why failures occurred, which is useful because it tells us which (temporarily) hard constraints should be made soft again. LCG is a hybrid approach to CP that uses a traditional ‘propagation and search’ constraint solver as the outer layer which guides the solution process, plus an inner layer which lazily decomposes CP to Boolean satisfiability (SAT) and applies learning SAT solver technology to reduce search [24,25].

The contributions of this paper are:

- We translate the basic unsatisfiable core approach of SAT to CP solving.
- We extend the basic unsatisfiable core approach to a nested version which more aggressively makes soft constraints hard.
- We discuss how we can use the unsatisfiable cores generated to improve the estimation of the objective function in CP search.
- We give experiments showing that for some CP optimization problems the unsatisfiable-core approach is significantly better than branch and bound.

## 2 Lazy Clause Generation

We give a brief description of propagation-based solving and LCG, for more details see [25]. We consider problems consisting of constraints  $\mathbf{C}$  over integer variables  $x_1, \dots, x_n$ , each with a given finite domain  $D_{\text{orig}}(x_i)$ . A feasible solution is a valuation  $\theta$  to the variables, which satisfies all constraints  $\mathbf{C}$ , and lies in the domain  $\mathbf{D}_{\text{orig}} = D_{\text{orig}}(x_1) \times \dots \times D_{\text{orig}}(x_n)$ , i.e.  $\theta(x_i) \in D_{\text{orig}}(x_i)$ .

A propagation solver maintains a domain restriction  $D(x_i) \subseteq D_{\text{orig}}(x_i)$  for each variable and considers only solutions that lie within  $\mathbf{D} = D(x_1) \times \dots \times D(x_n)$ . Solving interleaves propagation, which repeatedly applies propagators to remove unsupported values, and search which splits the domain of some variable and considers the resulting sub-problems. This continues until all variables are fixed (success) or failure is detected (backtrack and try another subproblem). A *singleton domain*  $D$  where all variables are fixed corresponds to a valuation  $\theta_D$  where  $\theta_D(x_i) = v_i$  when  $D(x_i) = \{v_i\}$ ,  $i \in 1..n$ .

Lazy clause generation is implemented by introducing Boolean variables for each potential value of a CP variable, named  $[x_i = j]$ , and for each bound,  $[x_i \geq j]$ . Negating them gives  $[x_i \neq j]$  and  $[x_i \leq j - 1]$ . Fixing such a *literal* modifies  $D(x_i)$  to make the corresponding fact true, and vice versa. Hence the literals give an alternate Boolean representation of the domain, which supports reasoning. Lazy clause generation makes use of *clauses* to record nogoods. A clause is a disjunction of literals, which we will often treat as a set of literals.

The high-level solving algorithm LCG, including propagation, search, and nogood generation, is shown as Algorithm 1. It is a standard CP branch-and-bound search, except that propagation (PROPAGATE) must return a nogood  $N$  as shown, explaining any failures that are detected by propagation. Propagation

---

**Algorithm 1** CP branch-and-bound with clause learning and backjumping

---

```
1: function LCG( $\mathbf{C}, \mathbf{D}, \mathbf{c}, \mathbf{y}$ )           % initial constraints, domains and objective
2:    $S \leftarrow []$                        % empty stack of domains per decision level
3:    $\theta \leftarrow \text{none}$                % best solution found so far, initially none
4:   loop
5:     % the call below updates the implication graph and  $N$ , not shown explicitly
6:      $\mathbf{D} \leftarrow \text{PROPAGATE}(\mathbf{C}, \mathbf{D})$ 
7:     if  $\mathbf{D} = \emptyset$  then
8:       % failure, nogood is  $N \rightarrow \text{false}$  where  $\text{decision\_level}(l) > 0$  for all  $l \in N$ 
9:       if  $N = \emptyset$  then               % conflict occurred at level 0
10:        return  $\theta$                      % no further improvement possible
11:      else
12:         $(L, M) \leftarrow \text{ANALYZE}(N)$  % make 1UIP nogood  $L \rightarrow l$  where  $M = \{l\}$ 
13:        pop  $S$  until reaching highest decision level of literals in  $L$  or 0
14:        pop  $\mathbf{D}$  from  $S$                    % backjump
15:         $\mathbf{C} \leftarrow \text{LEARN}(\mathbf{C}, L, M)$  % add redundant constraint to problem
16:      end if
17:      else if  $\mathbf{D}$  is a singleton domain then
18:        % found solution, record it and restart with tighter objective constraint
19:         $\theta \leftarrow \theta_D$ 
20:        pop  $S$  until reaching decision level 0
21:        pop  $\mathbf{D}$  from  $S$ 
22:         $\mathbf{C} \leftarrow \mathbf{C} \cup \{\mathbf{c}^T \mathbf{y} < \mathbf{c}^T \theta(\mathbf{y})\}$ 
23:      else
24:        % reached a fixed point, execute the user's programmed search strategy
25:        push  $\mathbf{D}$  onto  $S$ 
26:         $\mathbf{D} \leftarrow \text{DECIDE}(\mathbf{D})$ 
27:      end if
28:    end loop
29: end function

30: function ANALYZE( $N$ )
31:    $\text{conflict\_level} \leftarrow \max_{n \in N} \text{decision\_level}(n)$ 
32:   while there are multiple  $n \in N$  with  $\text{decision\_level}(n) = \text{conflict\_level}$  do
33:     let  $L \rightarrow l$  be the most recent unprocessed propagation at  $\text{conflict\_level}$ 
34:     if no such propagations remain unprocessed then break end if
35:     if  $l \in N$  then  $N \leftarrow (N - \{l\}) \cup L$  end if
36:   end while
37:   return  $(\{n : n \in N, \text{decision\_level}(n) < \text{conflict\_level}\},$ 
38:            $\{\neg n : n \in N, \text{decision\_level}(n) = \text{conflict\_level}\})$ 
39: end function

39: function LEARN( $\mathbf{C}, L, \{l\}$ )
40:   return  $\mathbf{C} \cup \{L \rightarrow l\}$ 
41: end function
```

---

must also record an *implication graph* showing the reasons for each propagation step, and for each literal which is fixed, its *decision level* as the value of  $|S|$  at the time of fixing. *Conflict analysis* derives new redundant constraints to avoid repeated search, and, as a side effect, modifies the backtracking procedure to *backjump* or restart solving at an appropriate point close to the failure [24].

The ANALYZE procedure reduces the information from the failure nogood  $N \rightarrow \text{false}$ , and the implication graph, into a 1UIP nogood, which can be learnt as a new redundant constraint. It considers propagations at the *conflict level*, which is the highest level of any literal in  $N$ . Working in reverse chronological order, for each propagation  $L \rightarrow l$ , where the propagated literal  $l$  occurs in  $N$ , this literal is replaced by its reason giving  $(N - \{l\}) \cup L$ . The process stops when there is at most one literal in  $N$  whose decision level is the conflict level, leaving a clause which propagates to fix that literal to its opposite value.

### 3 Soft Constraint Optimization

Soft constraints are constraints which should be respected if possible. When not all soft constraints can hold simultaneously we attach a *cost* to each violation. In the resulting optimization problem the overall cost is to be minimized. Soft constraints may be intensional or extensional. An intensional constraint is an equation or predicate capturing the desired relationship between variables, whereas an extensional constraint is written as a table with columns for the variables of interest, explicitly listing the allowed or disallowed tuples.

Specialized solvers have been highly successful for soft-constraint problems in extensional form. All of these solvers attempt to discover conflicts between soft constraints, or unsatisfiable soft constraints, by using what are essentially lookahead approaches, followed by appropriate reformulation that exposes the increased lower bound on solution cost due to the conflict or violation.

For WCSP, in which each extensional table contains a weight column giving the cost to be paid if the row holds, the best solver seems to be *tool-bar/toulbar2* [13,17]. It is based on a branch-and-bound search with consistency notions, where (loosely speaking) a variable is consistent if the *costs of the minimum cost value(s)* have been subtracted from the tables involving the variable and moved into the lower bound, thus *fathoming* unpromising branches.

For MAXSAT, good solvers in a recent evaluation [3] included *akmaxsat* [16] and *MaxSatz* [20] variants. Essentially they use lookahead, with unit propagation and failed literal detection, to improve the lower bound [18,19]. In restricted cases they use MAXSAT resolution, in which conflicting clauses are replaced by a unified clause plus compensation clauses [4]. WCSP solvers are also highly effective on MAXSAT, since MAXSAT is a special case of WCSP.

Recently there has also been considerable interest in decomposing MAXSAT to SAT, usually with an unsatisfiable-core approach [2,12,22,23]. Because they use learning instead of lookahead (and other improvements such as activity-based search [24]), they have a considerable advantage over the previously-described approaches. On the other hand they do not employ reformulation, and not all problems are suitable for unsatisfiable-core searches.

Pseudo-Boolean Optimization (PBO) is also promising for MAXSAT, which is a special case of PBO. In particular the Weighted Boolean Optimization (WBO) framework [21] is an application of PBO to soft-constraint problems, using some of the specialized techniques discussed above. Another option is decomposition to SAT via the PBO solver MiniSAT+ [9], which could be useful if unsatisfiability-based methods aren't applicable to a particular problem.

In this research we extend certain of the above techniques to *intensional* soft constraints. Modelling with intensional constraints has many advantages, (i) it is much easier since constraints are expressed in a natural way, (ii) it handles more constraints, since decomposition to extensional form is not always practical, and (iii) it can be more efficient, since propagation is a reasoning task as opposed to an expensive table traversal. It also has some disadvantages, (i) propagators must be implemented for each type of intensional constraint, and (ii) due to the many ways that constraints can interact, reformulation is difficult.

We consider solving combinatorial constrained optimization problems with *pseudo-Boolean objective* (COPPBO). A COPPBO  $(\mathbf{x}, \mathbf{y}, \mathbf{D}, \mathbf{C}, \mathbf{c})$  consists of a vector  $\mathbf{x}$  of general variables  $x_i$ ,  $i \in 1..m$ , a vector of  $\mathbf{y}$  of Boolean variables  $y_i$ ,  $i \in 1..n$  which appear in the objective, an initial finite domain  $\mathbf{D} = D(x_1) \times \dots \times D(x_m) \times D(y_1) \times \dots \times D(y_n)$ , a set  $\mathbf{C}$  of constraints  $C_i$ ,  $i \in 1..k$  and an objective  $z = \mathbf{c}^T \mathbf{y}$  to be minimized, where  $\mathbf{c}$  consists of positive constant weighting factors<sup>12</sup>. COPPBO problems encompass MAXSAT, partial MAXSAT, weighted partial MAXSAT, WBO and PBO.

An important class of COPPBO problems are *soft constraint optimization problems*  $(\mathbf{x}, \mathbf{D}, \mathbf{H}, \mathbf{S}, \mathbf{c})$  given by a set of hard constraints  $\mathbf{H}$  and a vector of soft constraints  $\mathbf{S}$ , with corresponding weight vector  $\mathbf{c}$  such that  $c_i$  is the cost of violating soft constraint  $S_i$ ,  $i \in 1..n$ . The aim is to find a solution  $\theta \in \mathbf{D}$  to the variables  $[\mathbf{x} \ \mathbf{y}]$ , which minimizes  $z = \mathbf{c}^T \mathbf{y}$ , subject to  $\mathbf{H} \cup \{\neg y_i \rightarrow S_i : i \in 1..n\}$ , where  $\mathbf{y}$  consists of introduced *relaxation variables* for the constraints in  $\mathbf{S}$ . Note that a CP system supporting constraint  $S_i$  can be straightforwardly extended to support the softened form  $\neg y_i \rightarrow S_i$  through half-reification [10].

## 4 Basic Unsatisfiable Cores Algorithm

An *unsatisfiable core* is a clause which contains only literals in  $\mathbf{y}$ . This clause forces an objective variable to be *true*, and must add some positive value to the objective. Note that clauses containing a literal  $\neg y_i$  are not unsatisfiable cores. The unsatisfiable cores approach to optimization originally arose for solving a MAXSAT problem, which is, given a set of soft clauses  $S$ , find a solution which satisfies the maximum number of soft clauses.

The basic unsatisfiable core solver consists of the procedures in Algorithm 2, which are called by the high level solver of Algorithm 1, and essentially modify the decision procedure based on information from conflict analysis. Each attempt fixes all *unfixed variables in  $\mathbf{y}$ , that have never appeared in an unsatisfiable core*,

<sup>1</sup> In calculating  $\mathbf{c}^T \mathbf{y}$  we take *false* = 0 and *true* = 1.

<sup>2</sup> We make the coefficients  $c_i$  positive by negating Boolean literals if necessary.

---

**Algorithm 2** Basic unsatisfiable core algorithm (relative to Algorithm 1)

---

**Require:**  $F = \{y_1, \dots, y_n\}$  initially

- 1: **function** LEARN( $\mathbf{C}, L, M$ ) where  $|M| > 1$  (otherwise fall back to Algorithm 1)
- 2:      $F \leftarrow F - M$      % remove candidates that have appeared in an unsatisfiable core
- 3:     **return**  $\mathbf{C}$      % unchanged constraint set, to avoid risk of learning duplicates
- 4: **end function**

**Require:**  $F$  = set of  $y_i$  literals which have never appeared in an unsatisfiable core

- 5: **function** DECIDE( $D(x_1) \times \dots \times D(x_m) \times D(y_1) \times \dots \times D(y_n)$ )
  - 6:     **if**  $decision\_level = 0$  and exists  $y_i \in F$  with  $D(y_i) = \{false, true\}$  **then**
  - 7:         for all such  $y_i$  restrict the corresponding domain  $D(y_i)$  to  $\{false\}$
  - 8:     **else**
  - 9:         restrict some other domain according to user's programmed search
  - 10:    **end if**
  - 11:    **return**  $D(x_1) \times \dots \times D(x_m) \times D(y_1) \times \dots \times D(y_n)$
  - 12: **end function**
- 

to *false*, and solves the resulting problem. This either finds a solution (which should be of low cost), or it detects that the problem is unsatisfiable. In a learning solver such as a SAT or LCG solver, by fixing the  $\mathbf{y}$ -variables as a (possibly) *multiple decision* in an artificial first decision level, failure occurring at this level generates, as a side effect, a new unsatisfiable core. This continues until solutions are found, or the original problem is proved unsatisfiable.

We have to modify the standard LCG solver to allow multiple simultaneous decisions when branching in procedure DECIDE. The ANALYZE procedure returns *generalized 1UIP nogoods*  $L \rightarrow M$  where  $L$  is a set of literals treated as a conjunction, and  $M$  a set treated as a disjunction, e.g.  $a \wedge b \rightarrow c \vee d$ . The code for ANALYZE in Algorithm 1 already handles this case, as line 34 will cause the loop to exit when only decisions remain. This line is unnecessary for search without multiple simultaneous decisions. Note also that, for the basic unsatisfiable core algorithm, we will only ever generate generalized 1UIP nogoods where  $L = \emptyset$ , but we will use the more general form in the next section.

Algorithm 2 keeps track of  $F$ , the set of optimization variables that have never appeared in an unsatisfiable core. The new post-analysis handler LEARN( $\mathbf{C}, L, M$ ) handles the case where  $|M| > 1$ , by removing variables in  $M$  from  $F$ . Unlike the 1UIP case with  $|M| = 1$ , it does not learn the new nogood  $L \rightarrow M$  since this may already be in the clause database, because two or more literals in  $M$  may have been set *false* simultaneously (propagating the database precludes single wrong decisions that violate a clause, but this does not extend to multiple decisions).

*Example 1.* The MAXSAT instance (all constraints soft) in the left column is entered as on the right, with objective  $z = y_1 + y_2 + y_3 + y_4$  to be minimized.

$$\begin{array}{ll|ll} C_1 \equiv \neg b & C_2 \equiv a \vee b & y_1 \vee \neg b & y_2 \vee a \vee b \\ C_3 \equiv \neg a & C_4 \equiv a & y_3 \vee \neg a & y_4 \vee a \end{array}$$

Here  $C_3$  and  $C_4$  are clearly in conflict, but given the choice it is better to relax  $C_3$  so that  $C_1$  and  $C_2$  can be satisfied simultaneously. At the top level the solver creates the multiple decision (simultaneously)  $y_1 = false, \dots, y_4 = false$  and

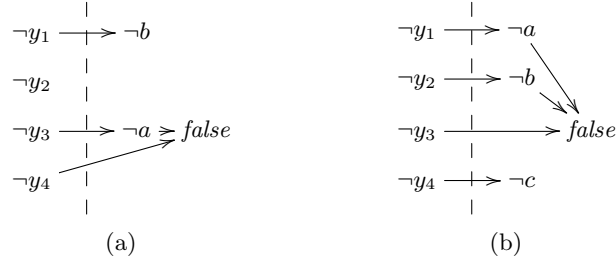


Fig. 1: Two implication graphs from solving the soft constraint problems of: (a) Example 1, and (b) Example 2; assuming all soft constraints are satisfied.

solves. This fails with unsatisfiable core  $y_3 \vee y_4$ , which is the generalized IUIP nogood resulting from the implication graph in Figure 1(a).

On the next attempt it sets only  $y_1 = false$ ,  $y_2 = false$ , because  $y_3, y_4$  have appeared in an unsatisfiable core. The only possible solution, under these assumptions, is  $b = false$ ,  $a = true$ ,  $y_3 = true$ ,  $y_4 = false$  with cost 1.

Although the initial solutions found by the unsatisfiable core algorithm are frequently of low cost, the algorithm does not replace the need for a traditional branch-and-bound approach (or equivalently the solving of a series of satisfaction problems with tighter and tighter bounds on the cost). When we have a (best known) solution  $\theta$ , the objective bound  $\mathbf{c}^T \mathbf{y} < \mathbf{c}^T \theta(\mathbf{y})$  needs to be propagated. This is standard for CP systems, although MAXSAT solvers typically use SAT decompositions of the objective using BDDs or sorting networks [22].

*Example 2.* Consider the trivial instance (original on left, reified on right),

$$\begin{array}{ll|ll} C_1 \equiv \neg a & C_2 \equiv \neg b & y_1 \vee \neg a & y_2 \vee \neg b \\ C_3 \equiv a \vee b & C_4 \equiv \neg c & y_3 \vee a \vee b & y_4 \vee \neg c \end{array}$$

The only unsatisfiable core is  $y_1 \vee y_2 \vee y_3$ , discovered from the implication graph in Figure 1(b). The solver will never try  $c = true$  because  $y_4$  does not participate in any unsatisfiable core. After discovering the core, search will set  $y_4 = false$  and propagate  $\neg c$ . Search will continue finding solutions with  $a = b = true$  at cost 2 or any other valuation to  $a, b$  at cost 1. Unsatisfiable-core algorithms do not distinguish between these cases.

Superficially the algorithm is a static search which fixes the variables in  $\mathbf{y}$  to *false* before proceeding, but is quite different because (i) any propagation resulting from fixing the  $\mathbf{y}$ -variables is done after *all* are fixed, which is enormously more efficient than one-by-one, especially when global constraints are involved, and (ii) static search backtracks only as far as necessary, so it will try all values for the tuple  $(y_1, \dots, y_n)$  in lexical order, which is impractical for large  $n$ .

In the best MAXSAT implementations [22,23] the relaxation variables for a clause were not created until the clause had appeared in an unsatisfiable subset, with the information about conflicting clauses being extracted from proof traces rather than from the presence of their relaxation literals in a learnt clause. This was sensible for MAXSAT considering that MAXSAT problems are ‘tall’,

i.e. they have more clauses than variables, hence more relaxation variables than ordinary variables, unless relaxation variables are created lazily.

On the other hand CP has much less reliance on decomposition due to its richer constraint library and problems are frequently ‘wide’, i.e. they have more variables than constraints, particularly when there is extensive use of global constraints. So relatively few relaxation literals are required and it was not worth going to the trouble of reformulating the problem ‘on the fly’, so we opted for the simpler approach of the artificial decision level, which allows conflict analysis to collect relaxation literals into unsatisfiable cores in a natural way.

## 5 Nested Unsatisfiable Core Algorithm

With the previously outlined approach to unsatisfiable cores it can happen that all or most variables in  $\mathbf{y}$  are covered by one or more unsatisfiable cores. This is particularly likely in problems with a lot of structure or symmetry such that each variable is involved in a similar set of constraints. In such cases, once all unsatisfiable cores have been enumerated, all soft constraints have reverted to being treated as soft, and the problem is no easier than originally.

Hence we define a nested version of the algorithm in which soft constraints can be made hard during search rather than only at the top level. This involves two new concepts, (i) *active* unsatisfiable cores which describe a conflicting set of soft constraints where we do not know which will be violated, as opposed to *inactive* where a particular violation is known and has already been penalized, and (ii) *contingent* unsatisfiable cores which are only unsatisfiable cores in the current subproblem, that is, under the current search assumptions.

Note that in the clausal view a *contingent* unsatisfiable core is simply any clause where only  $y_i$  literals remain, because other literals are already *false*. Then at least one of these  $y_i$  must be true, i.e. it is an unsatisfiable core. It is initially *active* but goes *inactive* when at least one of these  $y_i$  becomes *true*.

In the revised algorithm, an objective variable  $y_i$  may be set *false* initially by a multiple decision, but then reverts to unconstrained when search discovers it is part of an unsatisfiable core and backtracks. It will no longer be set to *false* since it appears in an active unsatisfiable core. But it may again be set *false* at a deeper decision level if all the unsatisfiable cores involving  $y_i$  go inactive.

*Example 3.* Referring to Example 2, given the unsatisfiable core  $y_1 \vee y_2 \vee y_3$ , on the next attempt  $y_4$  has not been involved in any core and so is set to *false* on level 1, hence  $c = \text{false}$ . On level 2, suppose search tries  $a = \text{true}$  hence  $y_1 = \text{true}$  and reaches a fixed point. Since  $y_1 \vee y_2 \vee y_3$  is now satisfied, there is no evidence that  $y_2$  or  $y_3$  needs to be *true*, and a new multiple decision is made as level 3 which sets  $y_2 = y_3 = \text{false}$ . Solving continues, finding  $b = \text{false}$  and cost 1.

*Example 4.* Consider minimizing  $a + b + c$  subject to the clauses

$$a \vee b \vee d \qquad b \vee c \vee \neg d \qquad \neg a \vee d,$$

which arises from the partial MAXSAT instance with soft constraints  $\neg a, \neg b, \neg c$  and hard constraints as above. Falsifying  $a, b, c$  returns the unsatisfiable core



---

**Algorithm 3** Nested unsatisfiable core algorithm (relative to Algorithm 1)

---

**Require:**  $a_i$  = count of *active* contingent unsatisfiable cores involving  $y_i$ , initially 0

**Require:**  $Y_i$  = stack of *all* contingent unsatisfiable core involving  $y_i$ , initially empty

**Require:** When literal  $y_i$  goes *true* due to propagation or decision,

1: **for**  $\{L \rightarrow M\}$  in stack  $Y_i$  where  $\{L \rightarrow M\}$  is still active **do**

2:     mark  $\{L \rightarrow M\}$  as inactive, and for all  $y_j \in M$  decrement  $a_j$

3: **end for**

4: **function** LEARN( $\mathbf{C}, L, M$ ) where  $|M| > 1$  (otherwise fall back to Algorithm 1)

5:     for all  $y_i \in M$  push the clause  $\{L \rightarrow M\}$  onto stack  $Y_i$  and increment  $a_i$

6:     **return**  $\mathbf{C}$      % unchanged constraint set, to avoid risk of learning duplicates

7: **end function**

8: **function** DECIDE( $D(x_1) \times \dots \times D(x_m) \times D(y_1) \times \dots \times D(y_n)$ )

9:     **if** exists  $y_i$  with  $a_i = 0$  and  $D(y_i) = \{false, true\}$  **then**

10:         for all such  $y_i$  restrict the corresponding domain  $D(y_i)$  to  $\{false\}$

11:     **else**

12:         restrict some other domain according to user's programmed search

13:     **end if**

14:     **return**  $D(x_1) \times \dots \times D(x_m) \times D(y_1) \times \dots \times D(y_n)$

15: **end function**

---

$a \vee b \vee c$ , so that on the next attempt, nothing can be fixed at the top level. This is exactly the problem that the nested algorithm tries to address.

Since a multiple decision is not applicable the ordinary decision procedure sets  $a = true$  on level 1, resulting in  $d = true$ , and reaches a fixed point. Because  $a \vee b \vee c$  is now satisfied, level 2 is a multiple decision setting  $b = c = false$ , but this immediately fails with reason  $b \vee c \vee \neg d$ , because  $d$  was already *true*. Solving backtracks past the multiple decision and records the new unsatisfiable core  $b \vee c$  (contingent on  $d = true$ ), then continues, using the ordinary decision procedure, eventually finding a solution of cost 2 (with  $a$  and either  $b$  or  $c = true$ ).

The benefit of this approach is that even if all objective literals eventually appear in an unsatisfiable core, we still aggressively set them to *false* as soon as all the active (contingent) unsatisfiable cores they appear in are satisfied. Conversely, it may be there are too many contingent unsatisfiable cores to be enumerated by this method, leading to thrashing in the solver.

The revised solver replaces Algorithm 2 with Algorithm 3 which maintains activity counts  $a_i$  for each variable  $y_i$ . When a new core becomes active through discovery in conflict analysis, the counts for all variables involved are incremented. When the core goes inactive, the counts are decremented, based on a cross-referencing data structure consisting of stacks  $Y_i$ . The counts  $a_i$  and stacks  $Y_i$  must be trailed so that they are reset correctly on backjumping. The DECIDE procedure only considers  $y_i$  variables with zero activity counts as candidates for a multiple decision, or if none exist it makes a normal search decision.

## 6 Notification-based Nested Algorithm

In the algorithms discussed to this point, we did not ‘learn’ unsatisfiable cores by adding them to the constraint store, because we did not want to risk learning duplicates, leading to duplicated propagation work. This is frustrating for the nested algorithm since contingent unsatisfiable cores are discarded on backtracking, even though they are valid nogoods which may capture new information.

*Example 5.* In Example 4, the first unsatisfiable core is  $a \vee b \vee c$  which holds new information not explicit in the original formulation, that can help future search. The next (contingent) unsatisfiable core is  $b \vee c \vee \neg d$  which is a duplicate of a clause in the original problem. Other than searching the constraint store or using a hashing scheme, we have no way of distinguishing between these cases.

Furthermore, with the previously-defined algorithms, clausal constraints from the constraint store are not used as contingent unsatisfiable cores except when they are violated by a multiple decision. To tackle this problem we modify the clausal propagators to issue a notification whenever an ordinary clause becomes an active contingent unsatisfiable core, that is, when the last of the non- $y_i$  literals in the clause goes *false*.

In essence we treat the clause  $C = X \vee Y$ , where  $Y$  are the  $y_i$  literals in  $C$  and  $X$  the others, as  $X \vee l$  and  $\neg l \vee Y$ , where  $l$  is a new literal. When  $l$  is set *true* by unit propagation, the clause  $C$  becomes a contingent unsatisfiable core. This is preferable because any multiple decisions made by the unsatisfiable core algorithms take into account all current information and do not lead to trivial failures. Then it is safe to learn all clauses resulting from conflict analysis.

*Example 6.* Revisiting Example 4, the first unsatisfiable core  $a \vee b \vee c$  has to be derived in the usual way by conflict analysis resulting from a multiple decision, but on the second attempt, as soon as  $d$  is set *true* by propagation from  $a$ , the *original* clause  $b \vee c \vee \neg d$  becomes an active contingent unsatisfiable core and is immediately recorded against the literals  $b$  and  $c$  as evidence that one of them must be *true*. This avoids the failed search with  $b = c = \text{false}$ .

## 7 Enhanced Lower Bounding

At each node of the branch and bound tree we need to check the *linear* constraint  $\mathbf{c}^T \mathbf{y} < \mathbf{c}^T \theta(\mathbf{y})$  where  $\theta$  is the current best solution and fathom (backtrack) when the constraint detects failure. This compares the current solution cost, or if we only have a partial solution then a conservative *lower bound* on the solution cost, with a conservative upper bound on the *optimal* solution cost. Usually the conservative lower bound is obtained by taking all unfixed  $y_i$  as *false*.

Information from unsatisfiable cores can be used to strengthen the lower bound. Suppose that some subset e.g.  $y_2$ ,  $y_3$  and  $y_5$  are unfixed but we have the unsatisfiable core  $y_2 \vee y_3 \vee y_5$ . Then clearly it is too conservative to assume that all are *false* and we can increase the estimate by at least  $\min(c_2, c_3, c_5)$ ,

---

**Algorithm 4** Disjoint unsatisfiable core-based bound-strengthening

---

```
function DISJOINT( $\mathbf{c}^T \mathbf{y} < u, \{G_1, \dots, G_\ell\}$ )
  for  $i$  in  $1..\ell$  do
     $\alpha_i \leftarrow \min_{y_j \in G_i} c_j$ 
    for all  $y_j$  in  $G_i$  do  $c_j \leftarrow c_j - \alpha_i$  end for
     $u \leftarrow u - \alpha_i$ 
  end for
  return  $\mathbf{c}^T \mathbf{y} < u$ 
end function
```

---

hopefully leading to earlier fathoming. This is the basis of *disjoint inconsistent subformula* approaches which have been used for MAXSAT [19].

*Example 7.* Referring to Example 2, after a finding the first solution of cost 1, the unsatisfiable core  $y_1 \vee y_2 \vee y_3$  implies that any solution will have cost  $\geq 1$  so no further improvement is possible. Solving can then terminate immediately.

Note that our treatment is somewhat more formal as LCG solvers require *reasons* for fathoming. Thus it is inadequate just to increase the estimate, we have to derive a globally true but tighter objective constraint. Suppose we have an objective constraint  $L \equiv \sum_{i=1}^n c_i y_i < u$  where  $u$  is the current upper bound, and a collection of *active* unsatisfiable cores  $C_1, \dots, C_\ell$ . Then considering core  $C_i$  as a *linear* constraint  $G_i \equiv \sum_{l \in C_i} -l \leq -1$ , we can produce a tightened objective constraint as a linear combination of  $L$  and  $G_i$ ,  $i \in 1..\ell$ .

*Example 8.* Suppose the problem is minimizing  $2y_1 + 3y_2 + 3y_3 + 5y_4$  with  $u = 7$  so that the objective constraint is  $U \equiv 2y_1 + 3y_2 + 3y_3 + 5y_4 < 7$ . Unsatisfiable core  $C_1 \equiv y_1 \vee y_3 \vee y_4$  gives the *linear* constraint  $G_1 \equiv -y_1 - y_3 - y_4 \leq -1$ . Then a strengthened objective constraint is obtained as the Fourier elimination [11] of  $y_1$ , by summing the constraints  $U' \equiv U + 2G_1 \equiv 3y_2 + y_3 + 3y_4 < 5$ . Unsatisfiable core  $C_2 \equiv y_2 \vee y_3 \vee y_4$  gives  $G_2 \equiv -y_2 - y_3 - y_4 \leq -1$  and eliminates  $y_3$  giving  $U'' \equiv U' + G_2 \equiv 2y_2 + 2y_4 < 4$ , which is tighter while still globally true.

We use two methods to strengthen the objective constraint  $U$ , (i) a heuristic method and (ii) an exact method. Note that the tightened inequality is contingent upon the non- $y_i$  literals of any contingent unsatisfiable cores that contributed to tightening, so whenever we have to explain the actions of the tightened propagator  $U$  we have to add these literals to the explanation.

**Disjoint Unsatisfiable Core-based Lower Bounding:** The heuristic method, shown as Algorithm 4, starts by taking a working version of the objective upper-bound constraint  $U$ . Then for each  $G_i$  it eliminates from  $U$ , by the Fourier-Motzkin method [11], the cheapest variable that could be made *true* to satisfy  $G_i$ , using the current working coefficients as costs. This means adding the assumed cost of satisfying  $G_i$  into the estimate, then adjusting the assumed costs used from then on, to avoid double-counting. Example 8 illustrates the algorithm.

The above algorithm is well known for weighted MAXSAT. We define an incremental version that modifies the objective constraint each time new active

unsatisfiable cores are discovered. When unsatisfiable cores go inactive they are removed from the tightened inequality, which is necessary for correctness, since the tightened bound constraint is a *consequence* of the original bound constraint, not equivalent. After removing an inactive core, we found it essential to re-examine subsequently-added cores to see if they can contribute any additional strength.

*Example 9.* Continuing Example 8, suppose later  $y_1 = \text{true}$ , then  $U''$  does not propagate, but  $U$  would have since it then becomes  $3y_2 + 3y_3 + 5y_4 < 5$  which immediately sets  $y_4 = \text{false}$ . We roll back to  $U$  where we eliminated the first core involving  $y_1$ , and now eliminate the remaining active core  $C_2 \equiv y_2 \vee y_3 \vee y_4$  to obtain  $U''' \equiv U + 3G_2 \equiv 2y_1 + 2y_4 < 4$ , which propagates  $y_4 = \text{false}$ .

**Linear Programming-based Lower Bounding:** The strengthening procedure is an optimization problem over the coefficients  $\alpha_i$  discussed above, which can be solved to optimality using a Linear Programming (LP) solver to give the best possible fathoming based on the information available at each node. The LP has to be updated at each node with the latest set of unsatisfiable cores, noting that inactive cores can be left in, since it can see they are already satisfied. The LP can also contain other constraints at the modeller’s discretion.

We define an LP constraint  $\text{linear\_program}(\mathbf{G}, \mathbf{c}, \mathbf{y}, z)$  which given a set of linear (in)equalities  $\mathbf{G}$  over the variables  $\mathbf{y}$ , an objective  $\mathbf{c}^T \mathbf{y}$  and an upper bound  $z$  on the objective (based on the best solution found so far), enforces that all (in)equalities are satisfied and  $\mathbf{c}^T \mathbf{y} \leq z$ . It executes when the bounds on  $\mathbf{y}$  are tightened, and verifies that  $\mathbf{G}$  can still hold, if not it detects failure with an explanation. It also verifies that there is still objective slack, if not it fathoms with an explanation. Then it prunes the  $\mathbf{y}$  if possible, based on the slack.

The *linear\_program* propagator works similarly to the bound-strengthening procedure described previously, in that it derives new *linear* constraints  $L + \sum_{i \in 1..l} \alpha_i G_i$ , choosing the vector  $\alpha$  which minimizes the RHS of the resulting constraint. For the LP case the vector  $\alpha$  and indeed the coefficients and RHS of the strengthened constraint are available directly in the dual solution after minimizing  $\mathbf{c}^T \mathbf{y}$  subject to  $\mathbf{G}$ . Thus the standard procedure of deriving explanations from dual solutions or unbounded dual rays is applicable [1,6,8].

## 8 Experiments

To evaluate whether unsatisfiable cores are useful for CP and whether our extensions are helping, we compared standard branch-and-bound with the solvers described in Sections 4, 5 and 6. With each of these solvers we tried standard lower bounding and each of the strengthened lower bounds from Section 7.

The basic solver is a state-of-the-art LCG solver, *Chuffed*. We use activity-based search (VSIDS) for all experiments [24]. For maximization problems we negate the objective, so all are minimization. The experiments were run on a cluster of AMD 6-core Opteron 4184 at 2.8 GHz with time limit 3600s, memory limit 2 Gbytes per core, and memory-outs treated as timeouts. Data files are available from [http://www.csse.unimelb.edu.au/~pjs/unsat\\_core](http://www.csse.unimelb.edu.au/~pjs/unsat_core).

Table 1: Evaluating unsatisfiable-core solvers on combinatorial benchmarks

		fathoming=std opt,s,sol,obj	disjoint opt,s,sol,obj	empty LP opt,s,sol,obj
PSM (k=1)	branch and bound	12,1117s, 16, -672	12, 1015s, 16, -674	11, 1214s, 16, -678
	basic unsat core	12,1084s, 16, -642	<b>12, 966s, 16, -643</b>	11, 1285s, 16, -641
	nested unsat core	8,1943s, 16, -629	12, 1461s, 16, -643	7, 2353s, 16, -599
	nested+notification	10,1658s, 16, -644	11, 1551s, 16, -648	12, 1341s, 16, -645
PSM (k=2)	branch and bound	2,3244s, 16, -654	1, 3375s, 14, -675	1, 3375s, 16, -724
	basic unsat core	0,3600s, 13, -581	1, 3375s, 13, -574	2, 3248s, 14, -567
	nested unsat core	0,3600s, 10, -559	2, 3161s, 13, -579	2, 3160s, 13, -596
	nested+notification	2,3167s, 16, -640	2, 3157s, 16, -626	<b>4,2973s, 16, -602</b>
PHOTO	branch and bound	10,2487s, 30, -15.7	10, 2492s, 30, -15.5	10, 2460s, 30, -15.6
	basic unsat core	11,2455s, 30, -15.9	14, 2211s, 30, -15.7	14, 2083s, 30, -15.8
	nested unsat core	10,2499s, 30, -15.8	17, 1781s, 30, -15.9	21, 1268s, 30, -15.9
	nested+notification	12,2437s, 30, -15.8	18, 1609s, 30, -15.9	<b>26, 926s, 30, -16.0</b>
RLFAP	branch and bound	1,3380s, 16, 11.0	1, 3379s, 16, 10.8	1, 3379s, 16, 10.8
	basic unsat core	2,3154s, 16, 9.7	<b>3,3086s, 16, 9.5</b>	2, 3154s, 16, 9.7
	nested unsat core	2,3156s, 14, 9.3	2, 3170s, 15, 9.1	1, 3375s, 14, 9.4
	nested+notification	2,3154s, 14, 9.2	2, 3174s, 15, 9.1	2, 3175s, 14, 9.2
ROSTER	branch and bound	14,1316s, 20, 15.7	14, 1297s, 20, 15.6	14, 1197s, 20, 16.4
	basic unsat core	16, 724s, 16, 15.4	16, 724s, 16, 15.4	16, 724s, 16, 15.4
	nested unsat core	16, 724s, 16, 15.4	16, 724s, 16, 15.4	16, 724s, 16, 15.4
	nested+notification	<b>16, 723s, 16, 15.4</b>	16, 725s, 16, 15.4	16, 724s, 16, 15.4
SUGIYAMA	branch and bound	5, 99s, 5, 8.6	5, 106s, 5, 8.6	5, 121s, 5, 8.6
	basic unsat core	5, 166s, 5, 8.6	5, 143s, 5, 8.6	5, 102s, 5, 8.6
	nested unsat core	5, 128s, 5, 8.6	5, 5s, 5, 8.6	5, 63s, 5, 8.6
	nested+notification	5, 207s, 5, 8.6	5, 14s, 5, 8.6	<b>5, 3s, 5, 8.6</b>

We tried the following combinatorial benchmarks: PSM (pattern set mining, 16 instances) is given a set of training items each a vector of Booleans, find some vector which characterizes all items, or if  $k > 1$  find the best  $k$  vectors [14]; PHOTO (30 instances) is given a set of people and soft constraints on who they stand next to, place them in a line for a photo; RLFAP (radio link frequency assignment, 16 instances) is assigning frequencies to channels with soft constraints that minimize interference [5]; ROSTER (20 instances) is finding a cyclic roster for a single worker over a number of weeks with soft work pattern constraints; and SUGIYAMA (5 instances) is a graph layout problem on layered graphs with soft no-edge-crossings constraints [27].

Table 1 shows for each solver: ‘opt’ number of instances optimized, ‘s’ mean solving time (taking the timeout for instances which timed out), ‘sol’ number of instances for which any solution was found, and ‘obj’ mean objective (taking the objective from the worst solver for instances where no objective is available, noting that this may be too generous). The solver which solves the most instances, falling back to comparing times and so on, is highlighted.

On these problems we see that with standard bound-estimation, we see that unsatisfiable cores is usually better than the unmodified solver but the improvement is not particularly dramatic. The best version of the unsatisfiable cores algorithm depends on the problem. Adding notification nearly always improves the nested algorithm, though SUGIYAMA is an exception which indicates an unusual constraint structure (perhaps there are too many unsatisfiable cores).

If the extra information from unsatisfiable cores is also used for bound-strengthening then dramatic improvements are possible. The best solver overall is the nested algorithm with notification and (initially empty) LP-based bound

Table 2: Evaluating unsatisfiable-core solvers on industrial problems

	fathoming=std opt,s,sol,obj,inf	empty LP opt,s,sol,obj,inf	redundant LP opt,s,sol,obj,inf	LP only opt,s,sol,obj,inf
CTT b&b	0,3600s,32, 66843,0	0,3600s,32, 69772,0	22, 1332s, 32, 82.9,0	13, 2734s, 32, 86.9,0
basic	0,3600s, 3, 62468,0	0,3600s, 1, 65611,0	24, 1166s, 28, 9349,0	8, 3059s, 10, 48750,0
nest	0,3600s, 1, 65485,0	0,3600s, 1, 65485,0	<b>24,1067s,29, 6045,0</b>	7, 3109s, 8, 57222,0
nest+not	0,3600s, 1, 65233,0	0,3600s, 1, 65233,0	24, 1107s, 29, 6045,0	9, 3094s, 9, 51673,0
STEIN b&b	0,3600s,13,265192,0	0,3600s,13, 9102,0	2, 3088s, 13, 9001,0	8, 1764s, 10, 8448,0
basic	0,3600s,13, 8432,0	3,2777s,13, 8533,0	4, 2540s, 13, 8433,0	11, 1041s, 13, 3520,0
nest	0,3600s,13, 7657,0	4,2498s,13, 8438,0	4, 2503s, 13, 6503,0	11, 944s, 13, 3425,0
nest+not	1,3443s,13, 6957,0	4,2495s,13, 6935,0	4, 2507s, 13, 5849,0	<b>12, 910s,13, 3422,0</b>
FCNF b&b	0,3600s, 2,188675,0	0,3600s,38,185854,0	9, 3176s,59,167292,0	25, 2290s, 47, 157774,0
basic	1,3556s,47,171065,0	0,3600s,48,169844,0	15, 2809s,60,134390,0	26, 2180s, 48, 153666,0
nest	1,3560s,40,183401,0	0,3600s,45,178483,0	16, 2722s,59,135616,0	23, 2295s, 50, 153992,0
nest+not	0,3600s,41,181917,0	7,3329s,46,177765,0	20, 2482s, 60,134105,0	<b>28,2009s,50,154185,0</b>
NSP b&b	3,3196s,29, 317,1	3,3177s,29, 319,1	2, 3162s, 30, 319,2	1, 3333s, 25, 344,2
basic	3,3263s,26, 278,0	8,2713s,26, 278,0	8, 2596s, 30, 272,2	8, 2537s, 28, 272,2
nest	3,3263s,27, 273,0	8,2702s,27, 273,0	7, 2591s, 30, 267,2	8, 2537s, 28, 269,2
nest+not	6,3009s,27, 274,0	8,2702s,27, 273,0	<b>8,2512s,29, 269,2</b>	8, 2519s, 27, 273,2

strengthening. We also see good results from the nested or basic algorithm and disjoint-based bound strengthening (without notification). The trade-off is that disjoint-based bounding is faster but sacrifices pruning power. Adding notification always helps LP-based strengthening, because the LP receives more information, but never helps disjoint-based strengthening, because overlapping unsatisfiable cores slow down the algorithm without adding strength.

We then tried some much more difficult industrial problems: CTT (curriculum-based timetabling, 32 instances) is finding a weekly repeating timetable for a university subject to various kinds of soft availability constraints and soft no-clashes constraints [7]; STEIN (Steiner network, 13 instances) is designing a connected network given a set nodes and arcs with a fixed (building) cost per arc [15]; FCNF (fixed-charge network flow, 60 instances) is designing a connected network with fixed (building) and also variable (operating) costs per arc [26]; and NSP (nurse scheduling problem, 32 instances) is designing a roster for a hospital ward based on the shift preferences of each individual nurse [29].

All of these problems use global constraints taken from *gcc*, *sequence*, *regular* and *network\_flow*. On CTT and NSP we utilize a hybrid search which considers each course/day sequentially, with activity-based search within courses/days.

Furthermore the objective in CTT and NSP is not pseudo-Boolean as some  $y_i$  are general integer. Then the unsatisfiable-cores algorithm expresses that e.g. all nurses should receive their first preference until we have evidence that this should be relaxed to the second preference for certain nurses and so on. We make the appropriate change to the multiple-decision algorithm, omitted earlier for clarity. Then unsatisfiable cores e.g.  $[y_1 \geq 3] \vee [y_2 \geq 2]$  say nurse 1 cannot receive 1st or 2nd preference simultaneously with nurse 2 receiving 1st preference. For bounding we add a linearization e.g.  $1/2(y_1 - 1) + (y_2 - 1) \geq 1$ , into the LP. We only consider LP-based bounding in this experiment, not disjoint-based.

These problems are extremely difficult to prove infeasible and so unsatisfiable-cores approaches were not highly successful initially ('std'/'empty LP' columns). We addressed this with a hybrid CP/MIP approach, by adding a *linear* decom-

Table 3: Comparing MIP solvers with enhanced CP solver

	<i>CPLEX</i> opt,s,sol,obj,inf	<i>SCIP</i> (learning) opt,s,sol,obj,inf	<i>Chuffed</i> (best solver) opt,s,sol,obj,inf
PSM (k=1)	9, 2061s, 16, -518,0	7,2514s,15, -447,0	<b>12, 966s,16, -643.0</b>
PSM (k=2)	1, 3375s, 13, -565,0	3,3240s, 8, -428,0	<b>4,2973s,16, -602.0</b>
PHOTO	3, 3488s, 30, -14.8,0	7,3216s,30, -15.4,0	<b>26, 926s,30, -16.0.0</b>
RLFAP	0, 3600s, 5, 10.4,0	1,3417s, 8, 10.2,0	<b>3,3086s,16, 9.5.0</b>
ROSTER	<b>18, 365s,20, 0.5,0</b>	17, 654s,20, 1.6,0	16, 723s, 16, 1.6,0
SUGIYAMA	3, 1887s, 5, 8.8,0	5, 816s, 5, 8.6,0	<b>5, 3s, 5, 8.6.0</b>
CTT	<b>30, 482s,32, 81.9,0</b>	21,1580s,32, 81.9,0	24, 1067s,29, 1474,0
STEIN	7, 1768s, 13, 3670,0	3,2770s,13, 3752,0	<b>12, 910s,13, 3422.0</b>
FCNF	<b>45,1042s,60,111387.0</b>	33,1702s,60,111864,0	28, 2009s,50,142562,0
NSP	<b>30, 63s,30, 263.2</b>	29, 190s,30, 264,2	8, 2512s,29, 267,2

position of all CP constraints, including globals, as a redundant *linear-program* propagator (‘redundant LP’ column). We also evaluated leaving out the original CP constraints and using only the LP, so that *Chuffed* becomes similar to a learning MIP solver such as *SCIP* using unsatisfiable cores (‘LP only’ column), but missing advanced MIP cutting planes and rounding heuristics.

Table 2 shows the results of these experiments. We add the column ‘inf’ as number of instances proved infeasible due to hard constraints. With the enhanced globality due to the *linear-program* propagator, the unsatisfiable-cores algorithm, in particular the nested algorithm with notification, is a big improvement for CP. On NSP (which has preferences), CP is useless without unsatisfiable cores.

Finally we consider MIP (without unsatisfiable cores) as an alternative to CP for solving soft-constraint problems. Table 3 compare CPLEX 12.0 and SCIP 2.1.1 with the best solver from each of the experiments described above. On the combinatorial benchmarks CP is clearly superior and much improved with our techniques. MIP was really only effective on the industrial problems. Given that timetabling and design problems require globality and that MIP employs many specialized and/or proprietary techniques, MIP should be the first resort for such problems. On the other hand CP with all our techniques is significantly improved and starts to look competitive, indeed on STEIN we beat MIP. On STEIN the ‘cuts’ added for LP-based bounding enforce the connectivity of the network and are similar to [15,26], but our approach is much more generic.

## 9 Conclusion

We have translated unsatisfiable-core methods from MAXSAT to solve CP optimization problems with pseudo-Boolean objectives, by making use of the facility of LCG solvers to generate unsatisfiable cores. This provides one of the first approaches to soft intensionally defined constraint problems beyond branch and bound that we are aware of, apart from PBO/WBO [9,21] which support intensionally-defined *linear* constraints only.

To gain the maximum advantage from unsatisfiable cores we needed to extend the method to generate and use unsatisfiable cores in the middle of search. This approach to optimization can be substantially better than the traditional branch and bound approach on intensionally defined optimization problems. With our extensions we saw a clear synergy between (i) aggressively assuming that soft

constraints hold, and (ii) using the resulting information about unsatisfiable or conflicting soft constraints, for enhanced fathoming.

## References

1. Achterberg, T.: Conflict analysis in mixed integer programming. *Discrete Optimization* 4(1), 4–20 (2007)
2. Ansótegui, C., Bonet, M., Levy, J.: Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In: Kullmann, O. (ed.) *Proc. SAT2009*, LNCS, vol. 5584, pp. 427–440. Springer Berlin / Heidelberg (2009)
3. Argelich, J., Li, C.M., Manyà, F., Planes, J.: Max-SAT 2011 Evaluation (2011), <http://www.maxsat.udl.cat/11>
4. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. *AI* 171(8-9), 606–618 (2007)
5. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.P.: Radio Link Frequency Assignment. *Constraints* 4, 79–89 (1999)
6. Davey, B., Boland, N., Stuckey, P.: Efficient Intelligent Backtracking Using Linear Programming. *IJOC* 14(4), 373–386 (2002)
7. Di Gaspero, L., Schaerf, A.: Neighborhood Portfolio Approach for Local Search Applied to Timetabling Problems. *JMMA* 5, 65–89 (2006)
8. Downing, N., Feydy, T., Stuckey, P.J.: Explaining Flow-Based Propagation. In: Beldiceanu, N., Jussien, N., Pinson, E. (eds.) *Proc. CPAIOR2012*. LNCS, vol. 7298, pp. 146–162. Springer Verlag, Nantes, France (June 2012)
9. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. *JSAT* 2(1-4), 1–26 (2006)
10. Feydy, T., Somogyi, Z., Stuckey, P.: Half reification and flattening. In: Lee, J. (ed.) *Proc. CP2011*, LNCS, vol. 6876, pp. 286–301. Springer Berlin / Heidelberg (2011)
11. Fourier, J.B.J.: Analyse des travaux de l’Académie Royale des Sciences, pendant l’année 1824; partie mathématique. In: *Histoire de l’Académie Royale des Sciences de France* 7, xlvii–lv (1827), partial English translation in D. A. Kohler, Translation of a report by Fourier on his work on linear inequalities, *Opsearch* 10: 38–42, 1973
12. Fu, Z., Malik, S.: On Solving the Partial MAX-SAT Problem. In: Biere, A., Gomes, C. (eds.) *Proc. SAT2006*, LNCS, vol. 4121, pp. 252–265. Springer Berlin / Heidelberg (2006)
13. de Givry, S., Heras, F., Zytnicki, M., Larrosa, J.: Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In: *Proc. IJCAI05*. pp. 193–198. AAAI Press (2005)
14. Guns, T., Nijssen, S., Raedt, L.D.: Itemset mining: A constraint programming perspective. *AI* 175(12-13), 1951–1983 (2011)
15. Koch, T., Martin, A.: Solving Steiner tree problems in graphs to optimality. *Networks* 32(3), 207–232 (1998)
16. Kuegel, A.: Improved Exact Solver for the Weighted MAX-SAT Problem. In: *Pragmatics of SAT workshop, SAT2010* (2010)
17. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for Weighted CSP. In: *Proc. IJCAI03*. pp. 239–244. AAAI Press (2003)
18. Li, C., Manyà, F., Planes, J.: Exploiting Unit Propagation to Compute Lower Bounds in Branch and Bound Max-SAT Solvers. In: van Beek, P. (ed.) *Proc. CP2005*, LNCS, vol. 3709, pp. 403–414. Springer Berlin / Heidelberg (2005)
19. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In: *Proc. AAAI06*. pp. 86–91. AAAI Press (2006)



20. Lin, H., Su, K., Li, C.M., Argelich, J.: IncWMaxSatz. In: Max-SAT 2008 Evaluation (2008), <http://www.maxsat.udl.cat/08>
21. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for Weighted Boolean Optimization. In: Kullmann, O. (ed.) Proc. SAT2009, LNCS, vol. 5584, pp. 495–508. Springer Berlin / Heidelberg (2009)
22. Marques-Silva, J., Planes, J.: Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In: Proc. DATE2008. pp. 408–413 (March 2008)
23. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. CoRR abs/0712.1097 (2007)
24. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proc. DAC2001. pp. 530–535 (2001)
25. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14, 357–391 (2009)
26. Ortega, F., Wolsey, L.A.: A branch-and-cut algorithm for the single-commodity, uncapacitated, fixed-charge network flow problem. Networks 41(3), 143–158 (2003)
27. Sugiyama, K., Tagawa, S., Toda, M.: Methods for Visual Understanding of Hierarchical System Structures. Systems, Man and Cybernetics, IEEE Trans. 11(2), 109–125 (Feb 1981)
28. Van Hoeve, W.J.: Operations Research Techniques in Constraint Programming. Ph.D. thesis, Centrum voor Wiskunde en Informatica, The Netherlands (2005)
29. Vanhoucke, M., Maenhout, B.: NSPLib – A Nurse Scheduling Problem Library: A tool to evaluate (meta-)heuristic procedures. In: Proc. ORAHS2005 (2005)